Bug Propagation through Code Cloning: An Empirical Study

Manishankar Mondal Chanchal K. Roy Kevin A. Schneider
Department of Computer Science and Engineering, University of Saskatchewan, Canada
{mshankar.mondal, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—Code clones are defined to be the identical or nearly similar code fragments in a code-base. According to a number of existing studies, code clones are directly related to bugs and inconsistencies in software systems. Code cloning (i.e., creating code clones) is suspected to propagate temporarily hidden bugs from one code fragment to another. However, there is no study on the intensity of bug-propagation through code cloning. In this paper we present our empirical study on bug-propagation through code cloning.**

We define two clone evolution patterns that reasonably indicate bug propagation through code cloning. We first identify code clones that experienced bug-fix changes by analyzing software evolution history, and then determine which of these code clones evolved following the bug propagation patterns. According to our study on thousands of commits of four open-source subject systems written in Java, up to 33% of the clone fragments that experience bug-fix changes can contain propagated bugs. Around 28.57% of the bug-fixes experienced by the code clones can occur for fixing propagated bugs. We also find that near-miss clones are primarily involved with bug-propagation rather than identical clones. The clone fragments involved with bug propagation are mostly method clones. Bug propagation is more likely to occur in the clone fragments that are created in the same commit operation rather than in different commits. Our findings are important for prioritizing code clones for refactoring and tracking from the perspective of bug propagation.

I. Introduction

Code cloning (i.e., copy/pasting) is a common yet controversial software engineering practice which is often employed by the programmers during development and maintenance for repeating common functionalities. Such a practice results the existence of identical or nearly similar code fragments, also known as code clones, in a code-base. Two code fragments that are similar to each other form a *clone-pair*. A group of similar code fragments forms a clone group or a *clone class*.

Code clones are of great importance from software maintenance perspectives. A great many studies [1], [2], [7], [8], [9], [14], [15], [17], [18], [19], [22], [23], [29], [30], [42], [20], [40], [12] have investigated the impacts of code clones on software evolution. While a number of studies [1], [8], [9], [15], [17], [18], [19] have identified some positive impacts of code clones, other studies [2], [14], [22], [7], [23], [29], [30], [20], [40], [12] have shown strong empirical evidence of negative impacts too. Code clones can be directly related to bugs and inconsistencies in the code-base [20], [21], [40]. It is commonly suspected that bugs can be propagated through code cloning. If a particular code fragment in a code-base contains a temporarily hidden bug, and a programmer copies that fragment to several other places being unaware of the presence of the bug, the bug in the original fragment gets propagated. Fig. 1 shows a possible way of bug-propagation through code cloning.

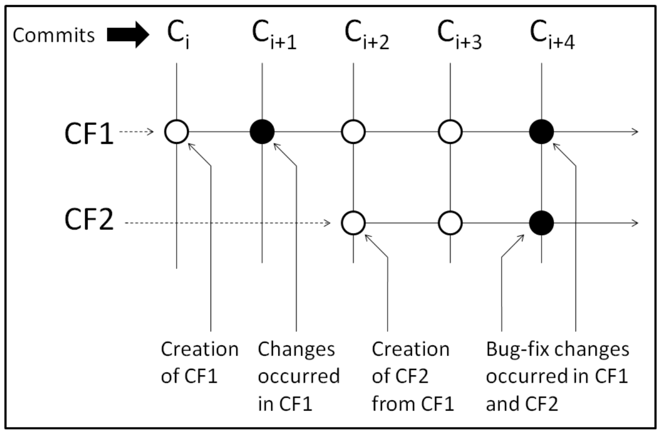


Fig. 1. A possible way of bug-propagation through code cloning

In Fig. 1, we can see the evolution of two clone fragments $CF1$ and $CF2$. As indicated in the figure, $CF1$ was created in commit operation C_i, and was changed in C_{i+1}. $CF2$ was created in commit C_{i+2} from $CF1$, and the two fragments, $CF1$ and $CF2$, made a clone-pair. In commit C_{i+4}, both of the clone fragments experienced a bug-fix change. Let us assume that the fragments were changed in the same way for bug-fixing. Thus, after experiencing bug-fixing changes, $CF1$ and $CF2$ remained as a clone-pair. From such a phenomenon we realize that $CF1$ contained a bug before $CF2$ was created from it. The bug might be introduced to $CF1$ at the time of its creation (i.e. in commit C_i) or in commit C_{i+1} (i.e., when $CF1$ was changed). However, the bug was not discovered just after being introduced. When $CF2$ was created from $CF1$ (in commit C_{i+2}), the bug in $CF1$ was propagated to $CF2$. Finally, in commit C_{i+4} the bug was fixed by making similar changes to $CF1$ and $CF2$.

Researchers suspect that code cloning can be responsible for propagating bugs. However, there is no study on how frequently bug-propagation occurs during code cloning. Without studying the intensity of bug-propagation we cannot properly realize the impacts of code clones on software evolution and maintenance. Focusing on this we perform an empirical study on bug-propagation in code clones. To the best of our knowledge, our study is the first one to investigate bug-propagation through code cloning. We have the following contributions:

- We define two patterns of bug propagation through code cloning.
- We propose an automatic mechanism of detecting these two bug propagation patterns. Our proposed mechanism works in two steps: (i) detecting bug-fix changes in code clones, and (ii) determining whether the bug-fix changes occurred for fixing propagated bugs by analyzing the evolution histories of the code fragments that experienced the bug-fix changes.

TABLE I. RESEARCH QUESTIONS

SL	Research Question
RQ 1	What percentage of code clones in different clone-types can be involved with bug propagation?
RQ 2	What percentage of the bugs that are experienced by different clone-types can be propagated bugs?
RQ 3	Which pattern of bug-propagation is more intense during evolution?

- We implement our bug propagation detection mechanism, apply it to four open-source subject systems written in Java, and investigate bug propagation through code cloning in these systems.

We answer three important research questions (Table I) regarding bug-propagation in code clones. According to our investigation on thousands of revisions of four subject systems:

- A considerable proportion of the code clones in a subject system can contain propagated bugs. We see that up to 33% of the code clones that experience bug-fixes can be involved with bug propagation.
- Near-miss clones (Type 2 and Type 3 clones) exhibit a higher intensity of bug-propagation compared to identical clones (Type 1 clones). Thus, near-miss clones should be given a higher priority for management from the perspective of bug-propagation.
- Around 28.57% of the bug-fix changes experienced by the code clones can occur for fixing propagated bugs.
- According to our manual analysis, the clone fragments that are involved with bug propagation are mostly method clones. Moreover, bug propagation primarily occurs to the clone fragments that are created together in the same revision (i.e., in the same commit operation). Thus, we suggest programmers to prioritize refactoring method clones that were created in the same revision.

We believe that bug-propagation should be taken into proper consideration when making clone management (such as clone refactoring or tracking) decisions. Our prototype tool that we have implemented for our study can be used for identifying code clones that are likely to contain propagated bugs. Thus, it can help programmers in managing code clones from the perspective of bug-propagation.

The rest of the paper is organized as follows. Section II contains the terminology, Section III discusses the experimental steps, Section IV defines the bug propagation pattern in code clones, Section V answers our research questions by presenting and analyzing our experimental results, Section VII discusses the related work, Section VIII mentions possible threats to validity, and Section IX concludes the paper by mentioning future work.

II. TERMINOLOGY

A. Types of Clones

We investigate bug propagation considering both exact (Type 1) and near-miss clones (Type 2 and Type 3 clones) [35], [34]. According to the literature, if two or more code fragments in a particular code-base are exactly the same disregarding the comments and indentations, these code fragments are called **exact clones** or **Type 1 clones** of one another. **Type 2 clones** are syntactically similar code fragments in a code-base. In general, **Type 2 clones** are created from Type 1 clones because of renaming identifiers and/or changing data types. **Type 3 clones** are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones. **Type 3 clones** are also known as *gapped clones*.

Clone-pair: If two code fragments in a particular code-base are similar to each other according to the above definitions of code similarity, we call them a clone-pair. A clone-pair can be of Type 1, Type 2, or Type 3.

B. Clone Fragment

We frequently use the term 'clone fragment' in our paper. A clone fragment is a particular code fragment which is exactly or nearly similar to one or more other code fragments in a code-base. Each member in a clone group or a clone-pair is a clone fragment.

C. Clone Genealogy

We detect clone genealogies for the purpose of our investigations. We define a clone genealogy in the following way. Let us assume that a clone fragment was created in a particular revision and was alive in a number of consecutive revisions. Thus, each of these revisions contains a snapshot of the clone fragment. The genealogy of this clone fragment consists of the set of its consecutive snapshots from the consecutive revisions where it was alive. Each clone fragment in a particular revision belongs to a particular clone genealogy. In other words, a particular clone fragment in a particular revision is actually a snapshot in a particular clone genealogy. By examining the genealogy of a clone fragment we can determine how it changed during evolution.

We automatically detect clone genealogies using the SPCP-Miner [25] tool. In our research, by examining the genealogy of a clone fragment we determine which commit operation(s) made changes to it.

D. Bug Propagation through Code Cloning

Existing studies [20], [21], [40] reveal that code clones can be related to bugs in software systems. It is also suspected that bugs in a code-base can get propagated through code cloning (copy/pasting). If a particular code fragment contains a bug which has not been discovered yet, then creating copies of that code fragment (i.e., cloning that code fragment) actually propagates the bug in all the created copies. If this bug gets discovered at a particular point of evolution, it should be fixed in each of the clone fragments that contains it. Thus, code cloning can increase bug-fixing effort during software evolution. Bug propagation tendencies of code clones should be considered when prioritizing them for management. Code clones with higher tendencies should be given higher priorities. Section IV discusses the details on bug propagation patterns. In our definition of bug propagation patters, we use the term *Similarity Preserving Co-change*. We discuss similarity preserving co-change in the following subsection.

E. Similarity Preserving Co-change of Clone Fragments

Mondal et al. [28] introduced the term *similarity preserving co-change*. We describe this in the following way. Let us consider two code fragments, CF1 and CF2, which are clones of each other in revision r of a subject system. A commit operation was applied on revision r and both of these two fragments were changed (i.e., the clone fragments co-changed) in such a way that they were again considered as

TABLE II. SUBJECT SYSTEMS

Systems	Lang.	Domains	LLR	Revs
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545

LLR = LOC in the Last Revision Revs = No. of Revisions

clones of each other in the next revision $r + 1$ (i.e., created because of the commit). In other words, the clone fragments preserved their similarity even after experiencing changes in the commit operation. Thus, this co-change of clone fragments (i.e., change of more than one clone fragment together) is a Similarity Preserving Co-change (SPCO).

Mondal et al. [28] showed that in a similarity preserving co-change (SPCO) more than one clone fragment from the same clone class are changed together consistently (i.e., the clone fragments are changed in the same way).

III. Experimental Steps

We perform our investigation on four Java systems listed in Table II. We download these systems from an on-line SVN repository [31]. We select our subject systems emphasizing their diversity in sizes and revision history lengths. We also see that the systems are of three different application domains.

A. Experimental Steps

We perform the following experimental steps before analyzing bug propagation in code clones: **(1)** Extraction of all revisions (as mentioned in Table II) of each of the subject systems from the online SVN repository; **(2)** Method detection and extraction from each of the revisions using CTAGS [6]; **(3)** Detection and extraction of code clones from each revision by applying the NiCad [5] clone detector; **(4)** Detection of changes between every two consecutive revisions using *diff*; **(5)** Locating these changes to the already detected methods as well as clones of the corresponding revisions; **(6)** Locating the code clones detected from each revision to the methods of that revision; **(7)** Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [23]; **(8)** Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy; **(9)** Detecting clone fragments that experienced bug-fix changes; and **(10)** Analyzing the evolution of bug-fix clones to identify bug propagation. For completing the first eight steps we use the tool SPCP-Miner [25]. In Section III-B we will discuss the technique that we apply for detecting bug-fix changes in code clones. We will define possible bug propagation patterns, and discuss how we detect such patterns in Section IV.

Clone Detection. We use the well-known NiCad [5] clone detector for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [37], [38]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of a minimum size of 10 LOC with 20% dissimilarity threshold and blind renaming of identifiers. For different settings of a clone detector the clone detection results can be different and thus, the experimental findings can also be different. For this reason selection of reasonable settings (i.e., detection parameters) is important. We used the mentioned settings in our research, because in a recent study [41] Svajlenko and Roy show that these settings provide us with better clone detection results in terms of both precision and recall. We should note that before using the NiCad outputs for Type 2 and Type 3 cases, we pre-processed them in the following way.

(1) Every Type 2 clone class that exactly matched any Type 1 clone class was excluded from Type 2 outputs.

(2) Every Type 3 clone class that exactly matched any Type 1 or Type 2 class was excluded from Type 3 outputs.

We performed these preprocessing steps because we wanted to investigate bug propagation in each of the three types of clones separately.

Clone Genealogies of Different Clone-Types. We use SPCP-Miner [25] to detect clone genealogies considering each clone-type (Type 1, Type 2, and Type 3) separately. Considering a particular clone-type this tool first detects all the clone fragments of that particular type from each of the revisions of the candidate system. Then, it performs origin analysis of these detected clone fragments and builds the genealogies. Thus, all the instances in a particular clone genealogy are of a particular clone-type. An instance is a snap-shot of a clone fragment in a particular revision. As we obtain three separate sets of clone genealogies for three different clone-types, we can easily determine the bug propagation intensities in these clone-types.

Tackling Clone-Mutations. Xie et al.[44] found that mutations of the clone fragments (i.e., a particular clone fragment may change its type) might occur during evolution. If a particular clone fragment is considered of a different clone-type during different periods of evolution, SPCP-Miner extracts a separate clone-genealogy for this fragment for each of these periods. Thus, even with the occurrences of clone-mutations, we can clearly distinguish which bugs were experienced by which clone-types.

B. Bug Detection in Code Clones

For a particular candidate system, we first retrieve the commit messages by applying the 'SVN log' command. A commit message describes the purpose of the corresponding commit operation. We automatically examine the commit messages using the heuristic proposed by Mockus and Votta [24] to identify those commits that occurred for the purpose of fixing bugs. Then we identify which of these bug-fix commits make changes to clone fragments. If one or more clone fragments are modified in a particular bug-fix commit, then it is an implication that the modification of those clone fragment(s) was necessary for fixing the corresponding bug. In other words, the clone fragment(s) are related to the bug. In this way we examine the commit operations of a candidate system, analyze the commit messages to retrieve the bug-fix commits, and identify those bug-fix commits that affected code clones.

The way we detect the bug-fix commits was also previously followed by Barbour et al. [2]. They detected bug-fix commits in order to investigate whether late propagation in clones is related to bugs. They at first identified the occurrences of late propagations and then analyzed whether the clone fragments that experienced late propagations are related to bug-fix. In our study we detect bug-fix commits in the same way as they detected, however, our study is not limited to late propagation clones only. We perform our study considering all clone fragments of a software system. Barbour et al. did not investigate bug-propagation. We investigate bug-propagation through code cloning in our study. Also, they did not consider Type 3 clones in their study. We consider Type 3 clones in our bug-propagation study.

After detecting bug-fix changes in code clones, we automatically detect whether the bug-fix clones evolved following a bug-propagation pattern.

IV. Bug Propagation Patterns

In the following two subsections we first provide formal definitions of two bug-propagation patterns, and then describe an automatic procedure that we have used for detecting these patterns. We should note that the two patterns we are going to describe include all possible ways of bug propagation through code cloning where the propagated bug was fixed in at least two clone fragments from the same clone class. Intuitively, a propagated bug should be fixed in the clone fragment that primarily contained the bug, and also, in the other clone fragments where the bug was propagated.

A. The First Bug Propagation Pattern

Propagation Pattern. Let us consider that two code fragments were created in a particular revision. These code fragments are similar to each other, and thus form a clone-pair. We also assume that a similar code fragment was not preexisting. As these two fragments were created in the same revision (i.e., in the same commit operation) it is likely that one fragment was first created by the programmer, and then she created the second one from the first one (possibly by copy/pasting). In this case, any bug that was introduced in the first fragment during its creation can be propagated to the second one. Considering such a way of bug-propagation we define the following bug-propagation pattern.

Pattern Definition. Let us consider that a clone-pair consisting of two clone fragments, CF1 and CF2, resides in revision r of a subject system. For fixing a bug, these clone fragments were changed together (i.e., were co-changed) in the commit operation c which was applied on revision r. We consider that this bug-fix change experienced by the two clone fragments in commit c is the fixing of a propagated bug if the following conditions hold:

- **Condition 1:** The two clone fragments, CF1 and CF2, were created in the same revision $r_{created}$. Here, $r_{created} < r$ (i.e., $r_{created}$ is older than r). No other similar code fragment was preexisting. In other words, a code fragment which is similar to CF1 and CF2 was not existing in revision $r_{created} - 1$. Here, $r_{created} - 1$ is the revision which was created just before the revision $r_{created}$.
- **Condition 2:** None or only one of the two clone fragments was changed during the evolution from $r_{created}$ to r. In other words, none or only one of the two fragments was changed in any of the commits that were applied on the revisions $r_{created}$ to $r - 1$. Here, $r - 1$ is the revision that was created just before revision r.
- **Condition 3:** The two clone fragments, CF1 and CF2, experienced a similarity preserving co-change in the bug-fix commit operation c which was applied on revision r. We have defined *similarity preserving co-change* in Section II. In a similarity preserving co-change, the clone fragments are updated consistently (i.e., the fragments are changed in the same way).

The first condition (*Condition 1*) implies the likeliness that one of the two fragments was created from the other one (possibly by copy/pasting). *Condition 2* confirms that after being created, at least one of the two fragments remained unchanged before they were both changed in the bug-fix commit c. *Condition 3* implies that each of the clone fragments was changed in the same way for fixing the bug, and thus, each fragment contained the same bug.

By analyzing the second condition (*Condition 2*) we realize that if none of the clone fragments got changed during the intermediate evolution (i.e., in the commits that were applied on the revisions $r_{created}$ to $r-1$), then the bug that was fixed in the two fragments in the commit operation c (the commit c was applied on revision r) was certainly introduced to them at the time of their creation (i.e., in the revision $r_{created}$). However, we see that *Condition 2* allows only one of the fragments to be changed in the intermediate evolution. Even with such a flexibility in the condition it is still confirmed that the bug that was fixed by consistently changing CF1 and CF2 in commit c was not introduced by any change during the intermediate evolution. Let us assume that the fragment CF2 was changed in a commit in the intermediate evolution. However, *Condition 3* confirms that both of the fragments were changed in the same way for bug-fixing in commit c. In other words, each of the changed fragment (i.e., CF2) and unchanged fragment (i.e., CF1) contained the same bug. Reasonably, any change in the intermediate evolution did not introduce the bug that was fixed in commit c, because CF1 also experienced the same fix even after remaining unchanged in the intermediate evolution. The bug was introduced (in one fragment) and propagated (to the other fragment) at the time of creation of the two fragments.

B. The Second Bug Propagation Pattern

Propagation Pattern. Let us assume that a code fragment CF2 was created in a particular revision from a similar preexisting code fragment CF1 (possibly by copy/pasting). Consequently, these two clone fragments made a clone-pair. In such a phenomenon it is possible that an unreported bug (i.e., a bug which has not yet been discovered) which was preexisting in the fragment CF1 will be transferred (i.e., propagated) to CF2. Considering this way of bug propagation we define the following bug propagation pattern.

Pattern Definition. Let us consider that two clone fragments, CF1 and CF2, make a clone-pair in revision r of a subject system. These two fragments were created at two different revisions: $r_{created1}$ and $r_{created2}$ respectively. Here, $r_{created1} < r$ and $r_{created2} < r$. In other words, both $r_{created1}$ and $r_{created2}$ are older than r. We also assume that $r_{created1} < r_{created2}$. Thus, CF1 was preexisting (i.e., CF1 is older than CF2). For fixing a bug, these two clone fragments were changed together (i.e., were co-changed) in the commit operation c which was applied on revision r. We consider that this bug-fix change experienced by the two fragments in commit c is the fixing of a propagated bug if the following three conditions hold:

- **Condition 1:** Just after the creation of the younger fragment (i.e., CF2), it was considered similar to the older one (i.e., CF1). Thus, CF1 and CF2 made a clone-pair from revision $r_{created2}$. We should note that $r_{created2}$ is older than r.

- **Condition 2:** None or only one of the two clone fragments was changed during the evolution period between the revisions $r_{created2}$ and r. In other words, none or only one of the fragments was changed in the commit operations applied on the revisions $r_{created2}$ to $r-1$. We have already mentioned that the revision r experienced the commit operation c which made changes to CF1 and CF2 for fixing a bug.
- **Condition 3:** The co-change of the two clone fragments (i.e., CF1 and CF2) in the commit operation c is a *similarity preserving co-change*.

As the clone fragments (CF1 and CF2) were created at two different commits, it implies the possibility that the younger code fragment CF2 might be created from the similar preexisting code fragment CF1. *Condition 1* implies that the bug that was fixed in commit c was introduced to CF2 at the time of its creation. *Condition 2* and *Condition 3* are similar to the corresponding conditions in the first pattern described in Section IV-A. This section (i.e., Section IV-A) also contains the discussions of these two conditions. Finally, the above three conditions reasonably imply the fixing of a bug that was preexisting in the older fragment CF1 and was propagated to the fragment CF2 through code cloning.

C. Automatic Detection of Bug Propagation Patterns

By following the procedure described in Section III-B we detect the bug-fix commits that affected code clones. Let us consider such a bug-fix commit which we call BFC. Let us further assume that a clone-pair was changed (i.e., both of the clone fragments in the pair were changed) in this commit for fixing a bug. We first determine whether the clone-pair experienced a similarity preserving co-change in the bug-fix commit BFC (i.e., we check **Condition 3** in each of the two patterns we have just described). If it did, then we analyze the genealogies of the two clone fragments in the pair. We previously mentioned that we use the tool SPCP-Miner [25] for detecting clone genealogies. By automatically examining the genealogies of the clone fragments we determine how they evolved in the past (i.e., how they evolved before the occurrence of the bug-fix commit BFC). We check each of the first two conditions of each of the two patterns by analyzing their genealogies. If no clone-pair experienced a similarity preserving co-change in the bug-fix commit BFC, then we consider that the clone related bug that was fixed in BFC is not a propagated bug.

D. An Example of Bug Propagation

In Fig. 2 we provide an example of fixing a propagated bug. The figure shows the similarity preserving co-change of two clone fragments (Clone Fragment 1, and Clone Fragment 2) in a bug-fix commit operation which was applied on revision 1349 of our candidate system Jabref. Our implemented prototype tool detects the evolution pattern of these two clone fragments as a bug-propagation pattern of the first category (i.e., defined in Section IV-A). Each of these two clone fragments were created in revision 1344. We see that the clone fragments have almost the same implementation except the condition parts of their *if-statements*. Clone Fragment 1 contains a NOT operator (!) which is not present in Clone Fragment 2. In such a phenomenon it is likely that one of these fragments was created from the other (possibly by copy/pasting), and thus the bug that was introduced in one fragment propagated to the other one. After being created in revision 1344, each of the fragments remained unchanged up to revision 1349. The commit operation which was applied on revision 1349 fixed the bug in these two clone fragments. From Fig. 2 and its caption it is clear that each of the fragments contained the same bug, and the fragments experienced a similarity preserving co-change for the purpose of bug-fixing. The changes that occurred to the clone fragments are also highlighted in the figure.

V. Experimental Results and Analysis

We apply our experimental steps on each of our subject systems and identify bug propagation patterns in each of the three types of code clones (Type 1, Type 2, and Type 3) separately. In the following subsections we answer the research questions by presenting and analyzing our experimental results.

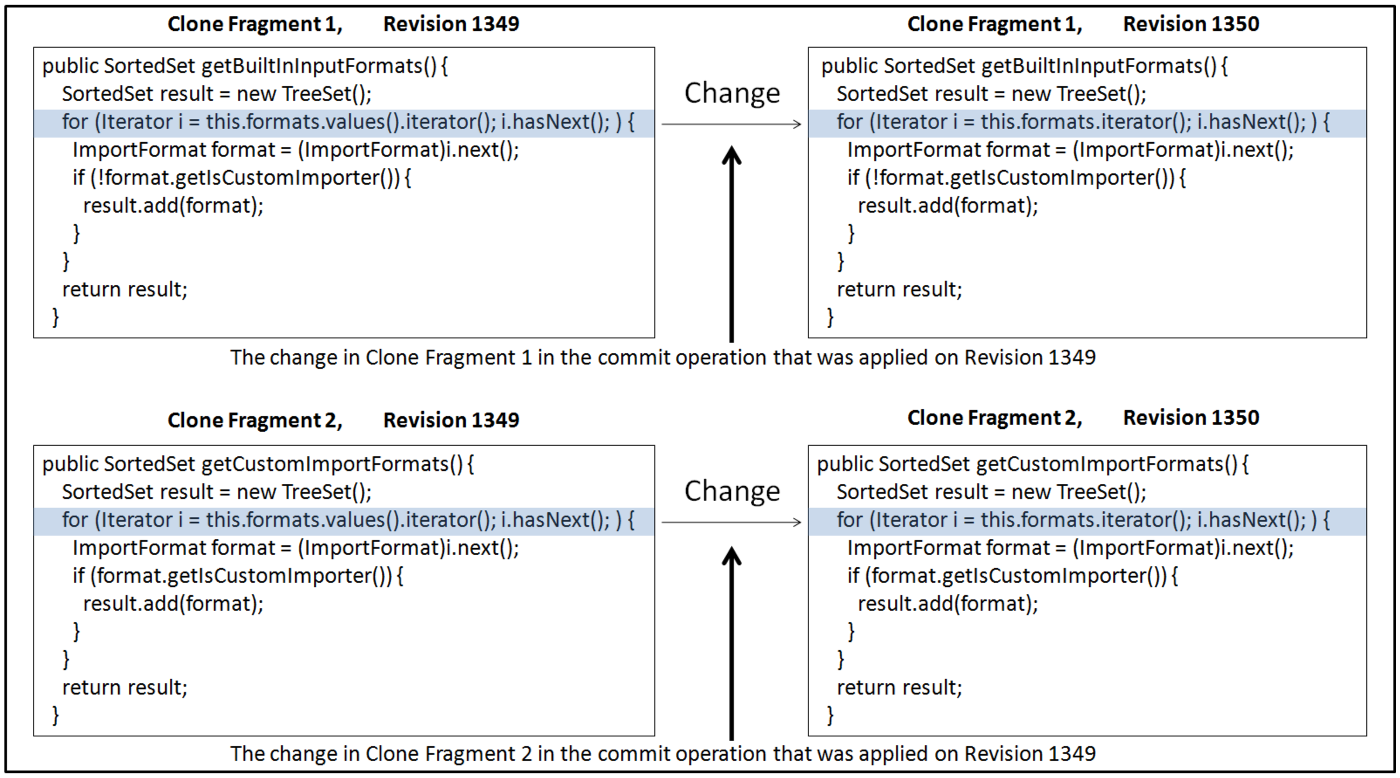
A. Answering RQ 1

RQ 1: *What percentage of code clones in different clone-types can be involved with bug propagation?*

Rationale. Answering RQ 1 is important. Bug propagation has always been considered a negative impact of code cloning. However, none of the existing studies investigated the intensity of bug propagation in code clones. Without investigating bug propagation intensities in different clone-types, we cannot properly realize the impact of code cloning on software evolution and maintenance. In RQ 1 we determine bug propagation intensity in each of the three major clone-types (Type 1, Type 2, and Type 3), and then make a comparative analysis of the intensities to determine which clone-type exhibits the highest intensity. We perform our investigation in the following way.

Methodology: For a particular subject system we determine all the bug-fix commits by applying the procedure described in Section III-B. Considering each clone-type we select those bug-fix commits where code clones of that particular type were changed for fixing bugs. For such a bug-fix commit for a particular clone-type, we determine whether a clone-pair has been changed (i.e., whether the two clone fragments in a clone-pair have been changed together or co-changed) in the commit. Considering each of the clone-pairs that have been changed in the bug-fix commit we automatically determine whether the two clone fragments in the pair evolved by following a bug propagation pattern defined in Section IV. In Section IV-C we described an automatic procedure for detecting a bug propagation pattern. If a clone-pair evolved by following a bug propagation pattern, we call this pair a *bug propagation clone pair*. Considering all the bug-fix commits affecting a particular clone-type we determine all the bug propagation clone pairs. For a particular clone-type of a subject system we determine the following measures, and report these in Table III.

- **CF (Clone Fragment):** The total number of distinct clone fragments (i.e., of the particular clone-type) that were created during the whole period of evolution of the subject system. This number is actually the total number of clone genealogies during system evolution.
- **BC (Bug-fix Commit):** The total number of bug-fix commits that affected clone fragments of that particular clone-type.
- **BCF (Bug-fix Clone Fragment):** The total number of clone fragments that experienced changes in the bug-fix commits. A particular clone fragment might


Clone Fragment 1, Revision 1349
Clone Fragment 1, Revision 1350
Change
The change in Clone Fragment 1 in the commit operation that was applied on Revision 1349
Clone Fragment 2, Revision 1349
Clone Fragment 2, Revision 1350
Change
The change in Clone Fragment 2 in the commit operation that was applied on Revision 1349

```
public SortedSet getBuiltInInputFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.values().iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (!format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

```
public SortedSet getBuiltInInputFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (!format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

```
public SortedSet getCustomImportFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.values().iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

```
public SortedSet getCustomImportFormats() {
  SortedSet result = new TreeSet();
  for (Iterator i = this.formats.iterator(); i.hasNext(); ) {
    ImportFormat format = (ImportFormat)i.next();
    if (format.getIsCustomImporter()) {
      result.add(format);
    }
  }
  return result;
}
```

Fig. 2. The figure shows a similarity preserving co-change of two clone fragments (Clone Fragment 1, and Clone Fragment 2) in the commit operation which was applied on revision 1349 of our subject system Jabref. The commit operation that was applied on revision 1349 is a bug-fix commit. The commit message says 'JabRef 2.0: fixed some Bugs'. We provide the snapshots of the two clone fragments in two revisions, 1349 and 1350, and highlight the differences between the snapshots. From the figure it is clear that the two clone fragments contained the same bug, and the fragments were changed in the same way for fixing the bug. This bug is a propagated bug, because the clone fragments evolved by following the first bug-propagation pattern defined in Section IV. We see that the difference between the two clone fragments lies in the condition part of the *if-statement*. Clone Fragment 1 contains a NOT operator (!) which is absent in Clone Fragment 2.

experience changes in more than one bug-fix commit. We determine the number of distinct clone fragments that experienced bug-fix changes. By analyzing the genealogy of a clone fragment we can identify which bug-fix commits it was changed in.

- **BPCP (Bug Propagation Clone Pair):** The number of distinct bug propagation clone pairs (i.e., the clone pairs that evolved following a bug propagation pattern defined in Section IV).
- **BPCF (Bug Propagation Clone Fragment):** The number of distinct clone fragments involved in the bug propagation clone pairs. We can easily understand that this number is the total number of clone fragments that were involved with bug propagation.

For each clone-type of each of the subject systems, we also determine the following two percentages considering the above measures.

- **PCFBP (Percentage of Clone Fragments involved with Bug Propagation):** This is the percentage of clone fragments that are involved with bug propagation with respect to all clone fragments in the system. We determine this using the following equation.

$$PCFBP = \frac{BPCF \times 100}{CF} \quad (1)$$

The bar graph in Fig. 3 shows this percentage for each clone-type of each of the subject systems.

- **PBCFBP (Percentage of Bug-fix Clone Fragments involved with Bug Propagation):** This is the percentage of clone fragments that are involved with bug propagation with respect to all bug-fix clone fragments. We calculate this in the following way.

$$PBCFBP = \frac{BPCF \times 100}{BCF} \quad (2)$$

Fig. 4 shows this percentage for each clone-type of each of the candidate systems.

From Fig. 3 we realize that Type 1 clones have the lowest possibility of being involved with bug propagation. For two subject systems, jEdit and Jabref, Type 1 clone fragments were not at all involved with bug propagation. It seems that bug propagation mainly occurs in Type 2 and Type 3 clones. We also show the overall percentages for three clone-types considering all subject systems. According to the overall scenario, Type 2 and Type 3 clones have comparable probabilities of propagating bugs. Bug propagation is negligible in Type 1 clones. We also see that the percentages plotted in Fig. 3 are generally very low. The reason behind this is that we calculate these percentages with respect to all clone fragments that were created during evolution. We know that a significant proportion of the code clones do not get changed at all during evolution [16]. The percentage of code clones that experience bug-fixes is generally very low (up to 19% according to a previous study [26]). The percentage of clone fragments that are involved with bug propagation should be even lower, because such

TABLE III. STATISTICS REGARDING BUG PROPAGATION IN DIFFERENT CLONE-TYPES

	Type 1					Type 2					Type 3				
Systems	**CF**	**BC**	**BCF**	**BPCP**	**BPCF**	**CF**	**BC**	**BCF**	**BPCP**	**BPCF**	**CF**	**BC**	**BCF**	**BPCP**	**BPCF**
Carol	415	8	31	1	2	211	8	32	2	4	682	22	134	130	39
Freecol	239	7	14	2	4	162	10	12	2	4	752	46	107	15	25
jEdit	7398	37	73	0	0	399	10	20	1	2	2688	42	184	4	8
Jabref	483	6	8	0	0	228	6	14	0	0	1363	23	31	1	2

CF = Total number of distinct clone fragments (i.e., clone genealogies) of a particular clone-type that were created during the whole period of evolution.
BC = Total number of bug-fix commits that affected clone fragments of a particular clone-type.
BCF = Total number of distinct clone fragments of a particular clone-type that experienced bug-fix commits (i.e., that experienced bug-fixes).
BPCP = Total number of distinct clone-pairs of a particular clone-type that evolved following a bug propagation pattern.
BPCF = Total number of distinct clone fragments of a particular clone-type that were evolved in the bug-propagation clone-pairs.

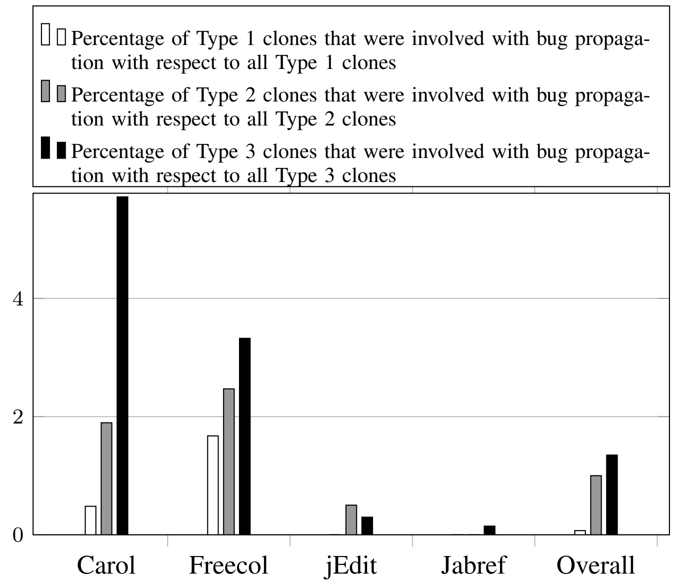


Fig. 3. Percentage of clone fragments that were involved with bug propagation with respect to all clone fragments considering each clone-type.

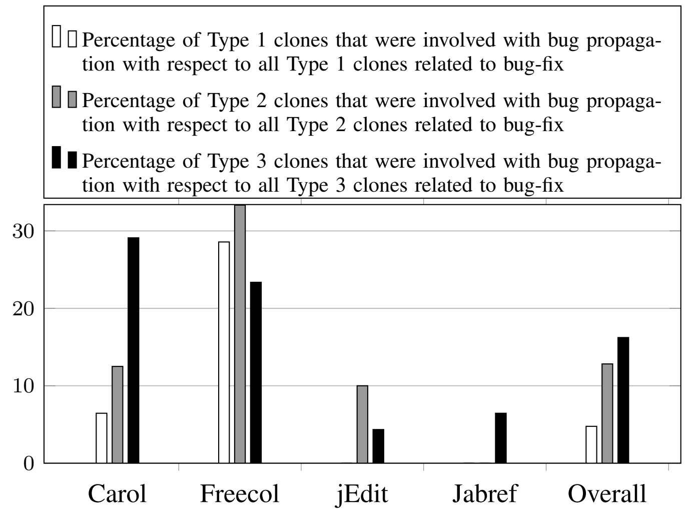


Fig. 4. Percentage of clone fragments that were involved with bug propagation with respect to all bug-fix clone fragments considering each clone-type.

clone fragments must be bug-fix clones and evolve following particular evolution patterns (defined in Section IV).

Fig. 4 shows the percentage of bug propagation clone fragments with respect to all bug-fix clones. We see that the percentages plotted in this graph are higher compared to the percentages plotted in the graph of Fig. 3. The overall comparative scenario of bug propagation in three clone-types presented in Fig. 4 is similar to that in Fig. 3. Fig. 4 also shows that up to 33% of the clone fragments that experience bug-fixing changes can be involved with bug propagation. The overall percentages of the bug-propagation clone fragments with respect to the bug-fix clones are 4.76%, 12.82%, and 16.22% for Type 1, Type 2, and Type 3 cases respectively.

Answer to RQ 1. From our experimental results and analysis we can state that *a considerable proportion of the clone fragments that experience bug-fixes can be involved with bug-propagation*. According to our subject systems, *up to 33% of the bug-fix clones contained propagated bugs. Type 1 clones exhibit the lowest possibility of being involved with bug propagation. Bug propagation is mainly observed in Type 2 and Type 3 clones with Type 3 clones showing the highest intensity of propagation.*

Our findings imply that the possibility of bug-propagation through exact copy/paste activities is very low, because identical clones have a very low possibility of containing propagated bugs. However, a considerable proportion of the near-miss clones can be involved with bug-propagation. Thus, near-miss clones should be considered more important for management (such as refactoring or tracking) than the identical clones from the perspective of bug-propagation. The prototype tool that we have implemented for our research can be customized for identifying code clones that are likely to contain hidden but propagated bugs. We believe that the comparative intensities of bug-propagation in different clone-types should be taken into proper consideration when making clone management decisions. Our prototype tool can help us make such decisions.

Manual Analysis of the Bug Propagation Patterns: We manually analyzed the evolution patterns of all the 133 bug propagation clone pairs (1 pair from Type 1 case + 2 pairs from Type 2 case + 130 pairs from Type 3 case) from our subject system Carol. We have the following observations from our manual analysis.

(1) For most of the bug propagation clone pairs (for 132 out 133), the two clone fragments in the pair did not get any change before experiencing the bug-fix change in the bug-fix commit. In other words, after getting created the first change that the two clone fragments experienced was the bug-fix change for most of the pairs. From the two patterns defined in Section IV we see that we allow only one of the two fragments to be changed before experiencing the bug-fix change. We found only one pair where one of the two fragments got changed before both experienced the bug-fix change.

(2) For most of the bug propagation clone pairs (for 131 out 133), the two clone fragments are full methods. It seems that bug propagation mainly occurs in method clones. For the remaining two pairs, the clone fragments were if-blocks and try-catch blocks respectively. Fig. 2 shows fixing of a propagated bug in two method clones. We provide this example from our subject system Jabref.

B. Answering RQ 2

RQ 2: *What percentage of the bugs that are experienced by different clone-types can be propagated bugs?*

Rationale. From our answer to the previous research question (i.e., RQ 1) we realize what proportions of the clone fragments in different clone-types can be involved with bug propagation. However, we still do not know what percentage of the bugs experienced by code clones can be propagated bugs. Without this information we cannot fully realize the bug propagation scenarios in different clone-types. In RQ 2 we first determine what percentage of the bugs experienced by each clone-type can be propagated bugs, and then make a comparison considering the percentages regarding three clone-types of each of the subject systems. We investigate in the following way.

Methodology. We first identify the bug-fix commit operations for a subject system following the procedure described in Section III-B. Considering a particular clone-type we select those commits where code clones of that particular type were changed. For such a commit we identify whether a clone-pair was changed (i.e., whether both of the clone fragments in the pair were co-changed) in it. Considering each such pair we determine whether the participating clone fragments evolved following a bug propagation pattern. The procedure for determining whether a clone-pair followed a bug propagation pattern has been discussed in Section IV-C. If a bug-fix commit modifies a clone-pair that evolved following a bug propagation pattern, we consider that the bug that was fixed in that commit is a propagated bug. We analyze each of the bug-fix commits that affected code clones of a particular clone-type and determine which bugs were propagated bugs. Considering each clone-type of each of the subject systems we determine the following two measures, and report these in Table IV:

- **BC (Bug-fix Commit):** The total number of bug-fix commits (i.e., the number of bugs) experienced by the code clones of that particular clone-type.
- **BCPB (Bug-fix Commit indicating fixing of a Propagated Bug):** The number of bug-fix commits that indicate fixing of propagated bugs.

For a particular clone-type of a particular subject system, we also determine the percentage of bug-fix commits that indicate fixing of a propagated bug (this percentage = $(BCPB \times 100)/BC$), and show this percentage in the bar graph of Fig. 5.

From Fig. 5 we see that none of the bug-fix commits in two subject systems, jEdit and Jabref, indicates fixing of a propagated bug in Type 1 clones. However, the percentage of bug-fix commits indicating fixing of propagated bugs is the highest (28.57%) in Type 1 case of our subject system Freecol. From Table IV we see that two out of seven bug-fix commits

TABLE IV. STATISTICS REGARDING THE NUMBER OF PROPAGATED BUGS IN DIFFERENT CLONE-TYPES

	Type 1		**Type 2**		**Type 3**	
Systems	**BC**	**BCPB**	**BC**	**BCPB**	**BC**	**BCPB**
Carol	8	1	8	2	22	4
Freecol	7	2	10	2	46	9
jEdit	37	0	10	1	42	3
Jabref	6	0	6	0	23	1

BC = Total number of bug-fix commits that affected clone fragments of a particular clone-type.
BCPB = The number of bug-fix commits that indicate fixing of a propagated bug.

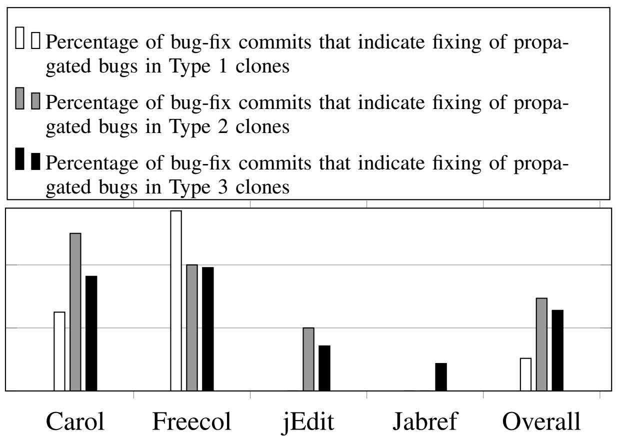


Fig. 5. Percentage of bug-fix commits that indicate fixing of propagated bugs in different clone-types

indicate fixing of propagated bugs. Looking at the data in this table it seems that bug-fix commits affecting Type 2 and Type 3 clones have higher possibilities of fixing propagated bugs compared to the bug-fix commits affecting Type 1 clones. We observe such a scenario from the overall percentages. We see that the overall percentages of bug-fix commits that fix propagated bugs in Type 2 and Type 3 clones are much higher compared to the percentage regarding Type 1 clones. Type 2 clones have the highest possibility of experiencing bug-fix commits that fix propagated bugs.

Answer to RQ 2: According to our experimental results and analysis, *a considerable proportion of the bugs experienced by code clones can be propagated bugs*. This percentage can be up to 28.57% according to our subject systems. *The overall percentage of propagated bugs is the highest in Type 2 case, and the lowest in Type 1 case*. However, this percentage regarding Type 3 case is also very near to that of Type 2 case.

Our findings from RQ 2 are similar to those from RQ 1. We find that near-miss clones (Type 2 and Type 3) have higher possibilities of containing propagated bugs compared to Type 1 clones. Thus, near-miss clones should be given a higher priority for management.

C. Answering RQ 3

RQ 3: *Which pattern of bug-propagation is more intense during evolution?*

Rationale. We have defined two bug propagation patterns in Section IV. It is important to investigate which pattern is more intense during evolution. If a particular pattern appears to be more intense than the other one, then we can prioritize

TABLE V. STATISTICS REGARDING THE NUMBER OF CLONE-PAIRS FOLLOWING DIFFERENT BUG PROPAGATION PATTERNS

	Type 1		Type 2		Type 3	
Systems	**PF**	**PS**	**PF**	**PS**	**PF**	**PS**
Carol	1	0	1	1	128	2
Freecol	2	0	1	1	6	9
jEdit	0	0	1	0	2	2
Jabref	0	0	0	0	1	0

PF = Number of clone pairs that followed the first pattern
PS = Number of clone pairs that followed the second pattern

refactoring of clone fragments that have evolved as well as that have the possibility of evolving following that pattern. We perform our investigation in the following way.

Methodology. Considering each clone-type of each of the subject systems we identify the bug-propagation clone-pairs as we did for answering our previous two research questions. Then, we determine the following two measures: (i) the number of clone-pairs that followed the first bug propagation pattern defined in Section IV-A, and (ii) the number of clone-pairs that followed the second bug propagation pattern defined in Section IV-B. These two measures for each clone-type of each of the subject systems have been reported in Table V. Using the data in Table V we also draw a stacked bar graph in Fig. 6 showing the percentage of bug propagation clone-pairs following each pattern.

From both Table V and Fig. 6 it is clear that bug propagation of the first category (defined in Section IV-A) is more likely to occur compared to the second one (defined in Section IV-B) during evolution. From the overall scenario (i.e., considering all subject systems) we realize that all the bug propagation clone pairs in Type 1 case (3 in total from Table V) followed the first pattern. For Type 2 case, 60% of the bug propagation clone pairs (i.e., 3 out of 5 pairs) followed the first pattern of propagation. At last, overall 91% of the bug propagation pairs in Type 3 case followed the first pattern.

Answer to RQ 3: According to our investigation, *the first bug propagation pattern where the two clone fragments in the bug propagation clone pair were created in the same revision is more likely to occur compared to the second pattern where the two clone fragments were created in two different revisions.*

Our finding implies that clone fragments that are created together in the same commit operation have higher possibilities of containing propagated bugs compared to the clone fragments that were created in different revisions. Thus, for minimizing bug propagation we prioritize refactoring of clone fragments that were created together.

VI. IMPLICATIONS FROM OUR FINDINGS

Although existing studies [30], [26], [33], [34] on code clone detection and analysis suspect that code cloning can be responsible for bug-propagation, none of the existing studies investigated it. Our study is the first one to show that bug-propagation is a fact. As bug-propagation occurs through code cloning, it is important to know which types of clones are more involved with bug-propagation so that programmers can avoid making such clones during programming. Also, if such clones are already existing in the code-base, we should consider refactoring or tracking them with high priority. Our answer to RQ 1 implies that near-miss clones are more involved with bug-propagation compared to exact clones. Thus, programmers should be careful when making near-miss clones. We should consider managing (refactoring or tracking) near-miss clones with higher priorities compared to exact clones. Our finding from RQ 3 implies that the first bug-propagation pattern is more frequent than the second one. Thus, making clones from an old code fragment is safer than making clones from a newly created code fragment. Intuitively, code fragments that served for a longer duration flawlessly should be considered for copying. The possibility that such fragments will contain bugs is lower compared to the newly created ones. Our manual analysis implies that method clones have high possibilities of being involved with bug-propagation. Thus, refactoring or tracking method clones can help us minimize bug-propagation.

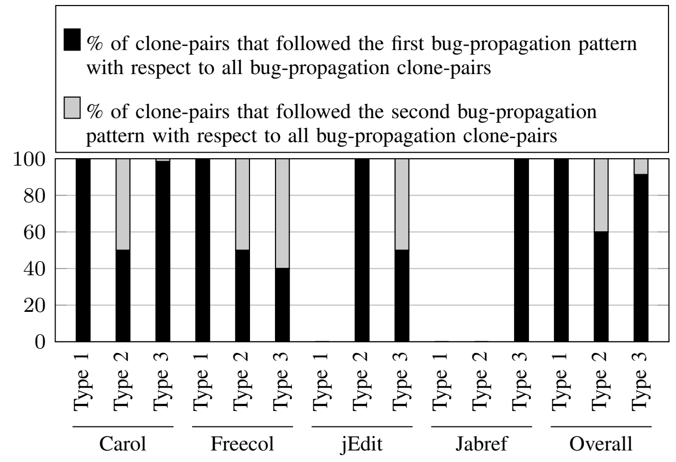


Fig. 6. Comparing the likeliness of occurrence of two bug propagation patterns.

VII. RELATED WORK

Bug-proneness of code clones has already been investigated by a number of studies. Li and Ernst [20] performed an empirical study on the bug-proneness of clones by investigating four software systems and developed a tool called CBCD on the basis of their findings. CBCD can detect clones of a given piece of buggy code. Li et al. [21] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [40] investigated on finding instances of incompletely fixed bugs in near-miss code clones by investigating a broad range of features of such clones involving machine learning. Göde and Koschke [7] investigated the occurrences of unintentional inconsistencies to the code clones of three mature software systems and found that around 14.8% of all changes occurred to the code clones are unintentionally inconsistent. Chatterji et al.[4] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al.[12] performed a study on the context based inconsistencies related to clones. They developed an algorithm to mine such inconsistencies for the purpose of locating bugs. Using their algorithm they could detect previously unknown bugs from two open-source subject systems. Inoue et al.[10] developed a tool called 'CloneInspector' in order to identify bugs related to inconsistent changes to the identifiers in the clone fragments. They applied their tool on a mobile software system and found a number of instances of such bugs.

Xie et al.[44] investigated fault-proneness of Type 3 clones in three open-source software systems. They investigated two evolutionary phenomena on clones: (1) mutation of the type of a clone fragment during evolution, and (2) migration of clone fragments across repositories and found that mutation of clone fragments to Type 2 or Type 3 clones is risky. We see that a number of studies investigated bug-proneness in code clones. However, none of these studies focus on bug propagation through code cloning.

Islam et al. [11] investigated bug-replication in code clones. They identified which of the clone fragments in a clone class contains the same bug. If more than one clone fragment in a clone class contain the same bug, they considered that the bug is a replicated one. However, it does not imply that this bug is a propagated bug. It might be the case that similar buggy changes occurred to the clone fragments in a clone class during evolution. Such a bug will be considered as a replicated bug according to Islam et al.'s [11] consideration. However, this is not a propagated bug. Bug propagation only occurs when a particular code fragment contains a bug, and this code fragment is copied to several other places in the code-base being unaware of the presence of the bug. In our study, we define two bug-propagation patterns and propose an automatic mechanism for identifying propagated bugs in code clones. Thus, our bug propagation study is significantly different from Islam et al.'s study [11].

In another study, Mondal et al. [26] compared the bug-proneness of three types of code clones. They investigated which types of code clones experience bug-fixes more frequently. However, they did not investigate bug-propagation in their study.

Rahman et al. [32] made a comparison of the bug-proneness of clone and non-clone code and found that clone code is less bug-prone than non-clone code. They performed their investigation on the evolution history of four subject systems using DECKARD [13] clone detector. However, they did not investigate bug-propagation through code cloning. Thus, our study on the intensity of bug-propagation in different types of code clones is different from their study.

Selim et al. [39] used Cox hazard models in order to assess the impacts of cloned code on software defects. They found that defect-proneness of code clones is system dependent. However, they considered only method clones in their study. We consider block clones in our study. While they investigated only two subject systems, we consider four subject systems in our investigation. Also, we investigate bug propagation in different types of code clones. Selim et al. [39] did not perform such an investigation.

A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al.[1] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al.[2], [3] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies.

We see that different studies have investigated clone related bugs in different ways and have developed different bug detection tools. However, none of these studies investigate the intensity of bug-propagation through code cloning. Investigating bug-propagation through code cloning is important. Without such an investigation we cannot properly realize the impacts of code cloning on software maintenance and evolution. Focusing on this issue we define and investigate two bug propagation patterns in code clones in our study. Our investigation involving manual analysis of the clone fragments that evolved following the bug-propagation patterns results interesting findings which are important for better management of code clones.

VIII. Threats to Validity

We used the NiCad clone detector [5] for detecting clones. For different settings of NiCad, the statistics that we present in this paper might be different. Wang et al. [43] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [36] and with these settings NiCad can detect clones with high precision and recall [37], [38], [41]. Thus, we believe that our findings on bug propagation through code cloning are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique followed by Barbour et al.[3]. Such a technique proposed by Mocus and Votta [24] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al.[3] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the comparative bug-proneness of clone-types. However, we selected our candidate systems emphasizing their diversity in sizes and revision history lengths. Thus, we believe that our findings cannot be attributed to a chance. Our findings are important from the perspectives of clone management and can help us in better ranking of code clones for refactoring and tracking.

IX. Conclusion

In this study we investigate the intensity of bug-propagation through code cloning. We define two bug-propagation patterns, and automatically mine these patterns by analyzing the entire evolution history of our subject systems. We perform our investigation on thousands of revisions of four subject systems and answered three important research questions. According to our analysis, up to 33% of the code clones that experience bug-fix changes can be related with bug-propagation. Near-miss clones (Type 2, and Type 3 cones) have a higher tendency of being involved with bug-propagation compared to identical clones (Type 1 clones). Thus, near-miss clones should be given a higher priority for management from the perspective of bug-propagation. We also observe that up to 28.57% of the bug-fix changes in code clones can occur for fixing propagated bugs. We manually investigate the occurrences of bug-propagation in code clones and discover that method clones are mostly involved with bug-propagation. Bug-propagation primarily occurs in the clone fragments that got created together in the same commit operation. Our prototype tool implemented for this study can assist programmers in identifying code clones that are likely to contain propagated bugs. Thus, our tool can be helpful for prioritizing code clones considering their likeliness of being involved with bug-propagation.

References

[1] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study", Proc. *CSMR*, 2007, pp. 81 – 90.

[2] L. Barbour, F. Khomh, Y. Zou, "Late Propagation in Software Clones", Proc. *ICSM*, 2011, pp. 273 – 282.

[3] L. Barbour, F. Khomh, Y. Zou, "An empirical study of faults in late propagation clone genealogies", *Journal of Software: Evolution and Process*, 2013, 25(11):1139 – 1165.

[4] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, N. A. Kraft, "Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study", Proc. *ESEM*, 2011, pp. 20 – 29.

[5] J. R. Cordy, C. K. Roy, "The NiCad Clone Detector", Proc. *ICPC Tool Demo*, 2011, pp. 219 – 220.

[6] CTAGS: http://ctags.sourceforge.net/

[7] N. Göde, Rainer Koschke, "Frequency and risks of changes to clones", Proc. *ICSE*, 2011, pp. 311 – 320.

[8] N. Göde, J. Harder, "Clone Stability", Proc. *CSMR*, 2011, pp. 65 – 74.

[9] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto, "Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software", Proc. *EVOL/IWPSE*, 2010, pp. 73 – 82.

[10] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, E. Lee, "Experience of Finding Inconsistently-Changed Bugs in Code Clones of Mobile Software", Proc. *IWSC*, 2012, pp. 94 – 95.

[11] J. F. Islam, M. Mondal, C. K. Roy, "Bug Replication in Code Clones: An Empirical Study", Proc. *SANER*, 2016, pp. 68 - 78.

[12] L. Jiang, Z. Su, E. Chiu, "Context-Based Detection of Clone-Related Bugs", Proc. *ESEC-FSE*, 2007, pp. 55 – 64.

[13] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", Proc. *ICSE*, 2007, pp. 96105.

[14] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, "Do Code Clones Matter?", Proc. *ICSE*, 2009, pp. 485 – 495.

[15] C. Kapser, M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software", *Empirical Software Engineering*, 2008, 13(6): 645 – 692.

[16] M. Kim, V. Sazawal, D. Notkin, G. C. Murphy, "An empirical study of code clone genealogies", Proc. *ESEC-FSE*, 2005, pp. 187 – 196.

[17] J. Krinke, "A study of consistent and inconsistent changes to code clones", Proc. *WCRE*, 2007, pp. 170 – 178.

[18] J. Krinke, "Is cloned code more stable than non-cloned code?", Proc. *SCAM*, 2008, pp. 57 – 66.

[19] J. Krinke, "Is Cloned Code older than Non-Cloned Code?", Proc. *IWSC*, 2011, pp. 28 – 33 .

[20] J. Li, M. D. Ernst, "CBCD: Cloned Buggy Code Detector", Proc. *ICSE*, 2012, pp. 310 – 320.

[21] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code", Proc. *OSDI*, 2004, pp. 20 – 20.

[22] A. Lozano, M. Wermelinger, "Tracking clones' imprint", Proc. *IWSC*, 2010, pp. 65 – 72.

[23] A. Lozano, M. Wermelinger, "Assessing the effect of clones on changeability", Proc. *ICSM*, 2008, pp. 227 – 236.

[24] A. Mockus, L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases", Proc. *ICSM*, 2000, pp. 120 – 130.

[25] M. Mondal, C. K. Roy, K. A. Schneider, "SPCP-Miner: A Tool for Mining Code Clones that are Important for Refactoring or Tracking", Proc. *SANER*, 2015, 5pp. (to appear).

[26] M. Mondal, C. K. Roy, K. A. Schneider, "A Comparative Study on the Bug-proneness of Different Types of Code Clones", Proc. *ICSME*, 2015, pp. 91 - 100.

[27] M. Mondal, C. K. Roy, K. A. Schneider, "Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems", Proc. *CASCON*, 2012, pp. 205 – 219.

[28] M. Mondal, C. K. Roy, K. A. Schneider, "Automatic Ranking of Clones for Refactoring through Mining Association Rules", Proc. *CSMR-WCRE*, 2014, pp. 114 – 123.

[29] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, K. A. Schneider, "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study", Proc. *SAC*, 2012, pp. 1227 – 1234.

[30] M. Mondal, C. K. Roy, K. A. Schneider, "An Empirical Study on Clone Stability", *ACM SIGAPP Applied Computing Review*, 2012, 12(3): 20 – 36.

[31] Online SVN repository: http://sourceforge.net/

[32] F. Rahman, C. Bird, P. Devanbu, "Clones: What is that Smell?", Proc. *MSR*, 2010, pp. 72 – 81.

[33] C. K. Roy, J. R. Cordy, "A Survey on Software Clone Detection Research", *Technical Report No. 2007-541*, 2007, School of Computing Queens University, pp. 1 - 115.

[34] C. K. Roy, M. F. Zibran, R. Koschke, "The Vision of Software Clone Management: Past, Present, and Future (Keynote paper)", Proc. *CSMR-WCRE*, 2014, pp. 18 – 33.

[35] C. K. Roy, "Detection and analysis of near-miss software clones", Proc. *ICSM*, 2009, pp. 447 – 450.

[36] C. K. Roy, J. R. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. *ICPC*, 2008, pp. 172 – 181.

[37] C. K. Roy, J. R. Cordy, R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *Science of Computer Programming*, 2009, 74 (2009): 470 – 495.

[38] C. K. Roy, J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools", Proc. *Mutation*, 2009, pp. 157 – 166.

[39] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, Y. Zou, "Studying the Impact of Clones on Software Defects", Proc. *WCRE*, 2010, pp. 13 - 21.

[40] D. Steidl, N. Göde, "Feature-Based Detection of Bugs in Clones", Proc. *IWSC*, 2013, pp. 76 – 82.

[41] J. Svajlenko, C. K. Roy, "Evaluating Modern Clone Detection Tools", Proc. *ICSME*, 2014, pp. 321 – 330.

[42] S. Thummalapenta, L. Cerulo, L. Aversano, M. D. Penta, "An empirical study on the maintenance of source code clones", *Empirical Software Engineering*, 2009, 15(1): 1 – 34.

[43] T. Wang, M. Harman, Y. Jia, J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation", Proc. *ESEC/SIGSOFT FSE*, 2013, pp. 455 – 465.

[44] S. Xie, F. Khomh, Y. Zou, "An Empirical Study of the Fault-Proneness of Clone Mutation and Clone Migration", Proc. *MSR*, 2013, pp. 149 – 158.